

```

/* modd.c

Copyright (c) 1993-2012. Free Software Foundation, Inc.

This file is part of GNU MCSim.

GNU MCSim is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 3
of the License, or (at your option) any later version.

GNU MCSim is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU MCSim; if not, see <http://www.gnu.org/licenses/>

-- Revisions -----
Logfile: %F%
Revision: %I%
Date: %G%
Modtime: %U%
Author: @a
-- SCCS -----

Contains routines for defining the model from the input file.
*/



#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <assert.h>

#include "lexerr.h"
#include "mod.h"
#include "lexfn.h"
#include "modd.h"
#include "modi.h"

/* List of valid functions that can be used in equations,
   they all return the type DOUBLE, except the BOOLEAN set.
*/
PSTR vrgszMathFuncs[] = {
    /* standard math functions */
    "acos", "asin", "atan", "atan2", "ceil", "cos", "cosh", "exp", "fabs",
    "floor", "fmod", "log", "log10", "pow", "sin", "sinh", "sqrt", "tan",
    "tanh",
}

```

```

/* special functions defined in random.c */
"CDFNormal", "erfc", "lnDFNormal", "lnGamma", "piecewise",

/* BOOLEAN functions used by SBML and included here for compatibility
*/
"and", "leq", "lt",

/* random sampling related routines defined in random.c */
"BetaRandom", "BinomialBetaRandom", "BinomialRandom", "CauchyRandom",
"Chi2Random",
"ExpRandom", "GammaRandom", "GetSeed", "GGammaRandom",
"InvGGammaRandom",
"LogNormalRandom", "LogUniformRandom", "NormalRandom",
"PiecewiseRandom",
"PoissonRandom", "SetSeed", "StudentTRandom", "TruncInvGGammaRandom",
"TruncLogNormalRandom", "TruncNormalRandom", "UniformRandom",

/* End flag */
"""

};

/* Global used by modd.c and modo.c as a flag */
char vszHasInitializer[] = "0.0; /* Redefined later */;

/* Functions */

/* -----
-----
*/
BOOL IsMathFunc (PSTR sz)
{
    int i = 0;

    while (*vrgszMathFuncs[i] && strcmp (vrgszMathFuncs[i], sz))
        i++;

    return (*vrgszMathFuncs[i]);
}

/* IsMathFunc */

/* -----
-----
*/
Check if the string passed corresponds to the CalcDelay function call.
If
    yes sets the global bDelays flag
*/
BOOL IsDelayFunc (BOOL *bDelays, PSTR sz)
{
    BOOL bIsDelay;

    bIsDelay = (!strcmp ("CalcDelay", sz));

```

```

    if (bIsDelay && !(*bDelays))
        *bDelays = TRUE;

    return (bIsDelay);

} /* IsDelayFunc */

/*
-----
*/
BOOL VerifyEqn (PINPUTBUF pibIn, PSTR szEqn)
{
    INPUTBUF ibDummy;
    PSTRLEX szLex;
    int iType, fContext;
    BOOL bReturn = TRUE;
    BOOL bOK = TRUE;
    PINPUTINFO pinfo;

    pinfo = (PINPUTINFO) pibIn->pInfo;

    MakeStringBuffer (pibIn, &ibDummy, szEqn);

    while (!EOB(&ibDummy)) { /* bOK not checked here... */

        NextLex (&ibDummy, szLex, &iType); /* ...all errors reported */

        switch (iType) {

            case LX_IDENTIFIER:
                if ((iType = GetKeywordCode (szLex, &fContext))) {
                    if (!(bOK = (iType == KM_DXDT && (fContext & pinfo-
>wContext))))
                        ReportError (pibIn, RE_BADCONTEXT | RE_FATAL, szLex, NULL);
                } /* if GetKeywordCode */
                else {
                    /* an input function cannot be assigned to something else than
an
                     input, so if szLex is recognized by GetFnType print an error
                     message */
                    if (GetFnType (szLex)) {
                        ReportError (pibIn, RE_BADCONTEXT | RE_FATAL, szLex, NULL);
                    }
                    /* Allowable identifiers are declared variables,
                     C functions, and the time variable */
                    if (!(bOK = (GetVarType (pinfo->pvmGloVars, szLex) ||
                                IsMathFunc (szLex) ||
                                IsDelayFunc (&(pinfo->bDelays), szLex) ||
                                ((pinfo->wContext == CN_DYNAMICS ||
                                  pinfo->wContext == CN_SCALE ||
                                  pinfo->wContext == CN_CALCOUPUTS) &&
                                !(strcmp(szLex, VSZ_TIME) &&
                                  strcmp(szLex, VSZ_TIME_SBML)))))))
                }
            }
        }
    }
}

```

```

        ReportError (pibIn, RE_UNDEFINED | RE_FATAL, szLex, NULL);
    }

    break;

case LX_EQNPUNCT:
    if ((szLex[0] == '!' || szLex[0] == '=') && strlen(szLex) == 1) {
        ReportError (pibIn, RE_UNEXPECTED, szLex, ".. in equation");
        bOK = FALSE;
    }
    break;

case LX_PUNCT:
    ReportError (pibIn, RE_UNEXPECTED, szLex, ".. in equation");
    bOK = FALSE;
    break;

case LX_INTEGER:
case LX_FLOAT:
    break;

default:
    ReportError (pibIn, RE_UNEXPECTED, szLex, ".. in equation");
    bOK = FALSE;
    break;

} /* switch */

bReturn = (bReturn && bOK);

} /* while */

return (bReturn);

} /* VerifyEqn */

```

---



---

AddEquation

Adds an equation to the list given. Note that this may be a variable table with initializer, or function defining equations.

The lists are maintained as stacks, i.e. Last one in is the head. This means that when writing equations to the output file, the pointers must first be reversed. I know it's not brilliant and is easily avoidable, but the simplicity of a stack was mighty attractive, and reversing pointers on a list of 20 or even 50 items is not a big deal considering the overall time savings will be minimal.

---

\*/  
void AddEquation (PVMMAPSTRCT \*ppvm, PSTR szName, PSTR szEqn, HANDLE hType)

```

{
    PVMMAPSTRCT pvmNew;

    if (!ppvm || !szName)
        return;

    if ((pvmNew = (PVMMAPSTRCT) malloc (sizeof(VMMAPSTRCT)))) {
        pvmNew->szName = CopyString (szName);
        pvmNew->szEqn = CopyString (szEqn);
        pvmNew->hType = hType;
        pvmNew->pvmNextVar = *ppvm;

        *ppvm = pvmNew; /* Redefine Head */
    } /* if */

    else
        ReportError (NULL, RE_OUTOFMEM | RE_FATAL, szName,
                     "* .. creating new equation in AddEquation");

} /* AddEquation */

/* -----
----- CopyString

Creates a buffer large enough to hold the given equation, copies it
into this buffer and returns a pointer to the buffer.

Reports memory errors.
*/
PSTR CopyString (PSTR szOrg)
{
    PSTR szBuf;

    if (szOrg) {
        if ((szBuf = (PSTR) malloc (strlen(szOrg) + 1)))
            return (strcpy ((PSTR) szBuf, szOrg));
        else
            ReportError (NULL, RE_OUTOFMEM | RE_FATAL, szOrg,
                         "* .. defining equation in CopyString");
    } /* if */

    return (NULL); /* else */

} /* CopyString */

/* -----
----- SetEquation

Sets the equation field of PVMMAPSTRCT given to the equation szEqn.
*/

```

```

void SetEquation (PVMMAPSTRCT pvm, PSTR szEqn)
{
    if (!pvm || !szEqn)
        return;

    if (pvm->szEqn)
        free (pvm->szEqn);

    pvm->szEqn = CopyString (szEqn);

} /* SetEquation */


/* -----
-----
SetVarType

Sets the iType field of the variable with szName. If szName is
not found, nothing is done without report of error. This routine
should be used for correcting known discrepancies, for instance
an initialization (to ID_PARM) before declaration of a model
variable.

*/
void SetVarType (PVMMAPSTRCT pvm, PSTR szName, HANDLE hType)
{
    while (pvm && strcmp (szName, pvm->szName))
        pvm = pvm->pvmNextVar;

    if (pvm)
        pvm->hType = hType;

} /* SetVarType */


/* -----
-----
GetVarPTR

Returns a pointer to the PVMMAPSTRCT structure of the variable
specified
by szName, or NULL if it does not exist.

*/
PVMMAPSTRCT GetVarPTR (PVMMAPSTRCT pvm, PSTR szName)
{
    while (pvm && strcmp (szName, pvm->szName))
        pvm = pvm->pvmNextVar;

    return (pvm);

} /* GetVarPTR */


/* -----
-----

```

```

GetVarType

Returns the code of the szName given. If the string does
not reference a declared variable, returns ID_NULL.

This function is also used as a test to prevent duplicate
declarations.

*/
int GetVarType (PVMMAPSTRCT pvm, PSTR szName)
{
    pvm = GetVarPTR (pvm, szName);
    return (TYPE(pvm));
}

} /* GetVarType */

/* -----
-----
CalculateVarHandle

Calculates what the handle created by the model generator will be.
This is kind of a sloppy routine, but it is needed so that the
input definitions can have parameter dependencies in the model
definition file.

The handle will be a bit combination of the type of variable, and
the index of the variable in the global map. Since the map is a
stack, we find the variable, and then get the index by counting
all of the variables of the same type that appear after it in the map.

This will give the index into the parm section of the map. The
routine AdjustVarHandles() in modo.c will do a fixup once all of
the variables are known.

*/
HANDLE CalculateVarHandle (PVMMAPSTRCT pvm, PSTR sz)
{
    PVMMAPSTRCT pvmVar;
    int cSameType = 0; /* Count of same type of variable in map */

    pvm = pvmVar = GetVarPTR (pvm, sz); /* Get PTR in map */

    if (pvm) /* Don't count the variable */
        pvm = pvm->pvmNextVar;

    while (pvm && pvm->hType == pvmVar->hType) {
        cSameType++; /* Count vars defined before it */
        pvm = pvm->pvmNextVar;
    } /* while */

    return ((HANDLE) (pvmVar ? pvmVar->hType | (HANDLE) cSameType : 0));
}

} /* CalculateVarHandle */

```

```

/* -----
DefineGlobalVar

Defines a global variable in the pvmGloVars list.
Redefinitions are reported, and ignored.
*/
void DefineGlobalVar (PINPUTBUF pibIn, PVMMAPSTRCT pvm,
                      PSTR szName, PSTR szEqn, HANDLE hType)
{
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmGloVars, szName, szEqn, ID_INLINE);

    else switch (hType) {

        case ID_NULL:
            AddEquation (&pinfo->pvmGloVars, szName, szEqn, ID_PARM);
            break;

        case ID_INPUT:
        case ID_OUTPUT:
        case ID_STATE:
            assert (pvm != NULL);
            if (!pvm->szEqn) { /* This is the first definition */
                if (hType == ID_INPUT) { /* Inputs use decl space for definition
*/
                    PIFN pifn = (PIFN) malloc (sizeof(IFN));

                    if (GetInputFn (pibIn, szEqn, pifn))
                        pvm->szEqn = (PSTR) pifn; /* THIS MAY CAUSE PROBS ON 68000 */
                    else
                        pvm->szEqn = NULL;
                } /* if */
            } /* else */
            else {
                pvm->szEqn = vszHasInitializer; /* Flag this variable */
                /* Add a new entry at end of list so
                   that dependencies are handled. */
                AddEquation (&pinfo->pvmGloVars, szName, szEqn, hType);
            } /* else */
            break;
        } /* if */

        /* else Redefinition -- Fall through ! */

        case ID_PARM:
            ReportError (pibIn, RE_REDEF | RE_WARNING, szName, NULL);
            break;

        default:
            ReportError (pibIn, RE_BADCONTEXT, szName, NULL);
    }
}

```

```

    } /* switch */

} /* DefineGlobalVar */

/* -----
----- DefineDynamicsEqn

Defines an equation in the pvmDynEqn list. State variable
assignments are meant as derivative assignments.

Redefinitions are reported, and ignored.
*/
void DefineDynamicsEqn (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, HANDLE
hType)
{
    HANDLE hNewType = (hType ? hType : ID_LOCALDYN);
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* Set ID_SPACEFLAG.
       This is a terrible kludge so we can use PVMMAPSTRCT struct. */
    hNewType |= ID_SPACEFLAG;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmDynEqns, szName, szEqn, ID_INLINE);

    else switch (hType) {

        case ID_NULL:
            AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);

            /* Fall through ! */

        case ID_LOCALDYN:
            AddEquation (&pinfo->pvmDynEqns, szName, szEqn, hNewType);
            break;

        case ID_FUNCTION:
            AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);
            break;

        case ID_DERIV: /* dt () assignment */
        case ID_STATE: /* Non-standard direct state assgn */
        case ID_OUTPUT:
            AddEquation (&pinfo->pvmDynEqns, szName, szEqn, hNewType);
            break;

        case ID_INPUT:
        case ID_PARM:
            ReportError (pibIn, RE_REDEF | RE_WARNING, szName,

```

```

        " Inputs and parameters cannot be assigned in
Dynamics\n");
        break;

    } /* switch */

    pibIn->iLNPrev = pibIn->iLineNum; /* Update prev eqn line num */

} /* DefineDynamicsEqn */

/* -----
-----
DefineScaleEqn

Defines an equation in the pvmScaleEqn list. All model variables
except inputs can be redefined here.
*/
void DefineScaleEqn (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, HANDLE
hType)
{
    HANDLE hNewType = (hType ? hType : ID_LOCALSCALE);
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* If there was more than one line since last equation, set
ID_SPACEFLAG.
   This is kind of a kludge so I can use PVMMAPSTRCT struct. */
    if (pibIn->iLineNum - pibIn->iLNPrev - 1)
        hNewType |= ID_SPACEFLAG;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmScaleEqns, szName, szEqn, ID_INLINE);

    else {

        if (!hType) /* Original type is NULL */
            AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);

        if ((hType & ID_LOCALSCALE) || /* Many per local */
            !GetVarPTR (pinfo->pvmScaleEqns, szName)) /* 1 eqn per parm */
            AddEquation (&pinfo->pvmScaleEqns, szName, szEqn, hNewType);

        else
            ReportError (pibIn, RE_REDEF | RE_WARNING, szName, "* Ignoring");
    }

    pibIn->iLNPrev = pibIn->iLineNum; /* Update prev eqn line num */

} /* DefineScaleEqn */

/* -----
-----
DefineCalcOutEqn
```

```

    Defines an equation in the pvmCalcOutEqn list. Only outputs
    can be redefined here.

*/
void DefineCalcOutEqn (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, HANDLE
hType)
{
    HANDLE hNewType = (hType ? hType : ID_LOCALCALCOUT);
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* If there was more than one line since last equation, set
ID_SPACEFLAG.
    This is kind of a kludge so I can use PVMMAPSTRCT struct. */
    if (pibIn->iLineNum - pibIn->iLNPrev - 1)
        hNewType |= ID_SPACEFLAG;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmCalcOutEqns, szName, szEqn, ID_INLINE);

    else {

        if (!hType) /* Original type is NULL */
            AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);

        /* FB 27 April 2013: allow redefinition of output variables */
        AddEquation (&pinfo->pvmCalcOutEqns, szName, szEqn, hNewType);

    }

    pibIn->iLNPrev = pibIn->iLineNum; /* Update prev eqn line num */

} /* DefineCalcOutEqn */

/*
-----
*/
void DefineJacobEqn (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, HANDLE
hType)
{
    HANDLE hNewType = (hType ? hType : ID_LOCALJACOB);
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* If there was more than one line since last equation, set
ID_SPACEFLAG.
    This is kind of a kludge so I can use PVMMAPSTRCT struct. */
    if (pibIn->iLineNum - pibIn->iLNPrev - 1)
        hNewType |= ID_SPACEFLAG;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmJacobEqns, szName, szEqn, ID_INLINE);

    else {

```

```

if (!hType) /* Original type is NULL */
    AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);

if ((hType & ID_LOCALJACOB) || /* Many per local */
    !GetVarPTR (pinfo->pvmJacobEqns, szName)) /* 1 eqn per parm */
    AddEquation (&pinfo->pvmJacobEqns, szName, szEqn, hNewType);

else
    ReportError (pibIn, RE_REDEF | RE_WARNING, szName, "* Ignoring");
}

pibIn->iLNPrev = pibIn->iLineNum; /* Update prev eqn line num */

} /* DefineJacobEqn */

/*
-----
*/
void DefineEventEqn (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, HANDLE
hType)
{
    HANDLE hNewType = (hType ? hType : ID_LOCALJACOB);
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* If there was more than one line since last equation, set
     ID_SPACEFLAG.
     This is kind of a kludge so I can use PVMMAPSTRCT struct. */
    if (pibIn->iLineNum - pibIn->iLNPrev - 1)
        hNewType |= ID_SPACEFLAG;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmEventEqns, szName, szEqn, ID_INLINE);

    else {

        if (!hType) /* Original type is NULL */
            AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);

        if ((hType & ID_LOCALJACOB) || /* Many per local */
            !GetVarPTR (pinfo->pvmEventEqns, szName)) /* 1 eqn per parm */
            AddEquation (&pinfo->pvmEventEqns, szName, szEqn, hNewType);

        else
            ReportError (pibIn, RE_REDEF | RE_WARNING, szName, "* Ignoring");
    }

    pibIn->iLNPrev = pibIn->iLineNum; /* Update prev eqn line num */

} /* DefineEventEqn */

/*
-----

```

```

/*
void DefineRootEqn (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, HANDLE
hType)
{
    HANDLE hNewType = (hType ? hType : ID_LOCALJACOB);
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* If there was more than one line since last equation, set
    ID_SPACEFLAG.
        This is kind of a kludge so I can use PVMMAPSTRCT struct. */
    if (pibIn->iLineNum - pibIn->iLNPrev - 1)
        hNewType |= ID_SPACEFLAG;

    if (!strcmp (szName, "Inline")) /* Inline statements are accumulated */
        AddEquation (&pinfo->pvmRootEqns, szName, szEqn, ID_INLINE);

    else {

        if (!hType) /* Original type is NULL */
            AddEquation (&pinfo->pvmGloVars, szName, NULL, hNewType);

        if ((hType & ID_LOCALJACOB) || /* Many per local */
            !GetVarPTR (&pinfo->pvmRootEqns, szName)) /* 1 eqn per parm */
            AddEquation (&pinfo->pvmRootEqns, szName, szEqn, hNewType);

        else
            ReportError (pibIn, RE_REDEF | RE_WARNING, szName, "* Ignoring");
    }

    pibIn->iLNPrev = pibIn->iLineNum; /* Update prev eqn line num */

} /* DefineRootEqn */

/*
-----
DefineVariable

Define the variable szLex according to szEqn if the variable
type is valid for the assignment in the given fContext.

* If iKWCode is KM_DXDT and the context is CN_DYNAMICS, defines a
  state equation for szName. States cannot be assigned direct
  values in the Dynamics section.

* Inputs cannot be assigned in the Dynamics section.

* Values given to States in a global context are initial values.

* In the global parameter declarations, a duplicate definition issues
  a warning and ignores the redefinitions
*/
void DefineVariable (PINPUTBUF pibIn, PSTR szName, PSTR szEqn, int
iKWCode)

```

```

{
    PVMMAPSTRCT pvm;
    HANDLE hGloVarType;
    PINPUTINFO pinfo;

    pinfo = (PINPUTINFO) pibIn->pInfo;

    assert (pinfo->wContext != CN_END);

    if (!szName || !szEqn)
        return;

    pvm = GetVarPTR (pinfo->pvmGloVars, szName);
    hGloVarType = TYPE(pvm);

    /* the variable has been found but was defined as a local for
       another section that the current one, it need to be redefined
       as local, so we undo the above assignments */
    if (((pinfo->wContext == CN_DYNAMICS) &&
        ((hGloVarType == ID_LOCALSCALE) ||
         (hGloVarType == ID_LOCALJACOB) ||
         (hGloVarType == ID_LOCALEVENT) ||
         (hGloVarType == ID_LOCALROOT) ||
         (hGloVarType == ID_LOCALCALCOUT))) ||
        ((pinfo->wContext == CN_SCALE) &&
         ((hGloVarType == ID_LOCALDYN) ||
          (hGloVarType == ID_LOCALJACOB) ||
          (hGloVarType == ID_LOCALEVENT) ||
          (hGloVarType == ID_LOCALROOT) ||
          (hGloVarType == ID_LOCALCALCOUT))) ||
        ((pinfo->wContext == CN_JACOB) &&
         ((hGloVarType == ID_LOCALDYN) ||
          (hGloVarType == ID_LOCALSCALE) ||
          (hGloVarType == ID_LOCALJACOB) ||
          (hGloVarType == ID_LOCALROOT) ||
          (hGloVarType == ID_LOCALCALCOUT))) ||
        ((pinfo->wContext == CN_EVENTS) &&
         ((hGloVarType == ID_LOCALDYN) ||
          (hGloVarType == ID_LOCALSCALE) ||
          (hGloVarType == ID_LOCALJACOB) ||
          (hGloVarType == ID_LOCALROOT) ||
          (hGloVarType == ID_LOCALCALCOUT))) ||
        ((pinfo->wContext == CN_ROOTS) &&
         ((hGloVarType == ID_LOCALDYN) ||
          (hGloVarType == ID_LOCALSCALE) ||
          (hGloVarType == ID_LOCALJACOB) ||
          (hGloVarType == ID_LOCALEVENT) ||
          (hGloVarType == ID_LOCALCALCOUT))) ||
        ((pinfo->wContext == CN_CALCOUTPUTS) &&

```

```

((hGloVarType == ID_LOCALDYN) ||  

(hGloVarType == ID_LOCALSCALE) ||  

(hGloVarType == ID_LOCALEVENT) ||  

(hGloVarType == ID_LOCALROOT) ||  

(hGloVarType == ID_LOCALJACOB))) {  

  

    pvm = NULL;  

    hGloVarType = TYPE(pvm);  

}  

  

if ((iKWCode != KM_INLINE) &&  

    (hGloVarType != ID_INPUT || pinfo->wContext != CN_GLOBAL) &&  

    !VerifyEqn (pibIn, szEqn))  

    return; /* Errors reported in Verify eqn */  

  

switch (pinfo->wContext) {  

    case CN_GLOBAL:  

        DefineGlobalVar (pibIn, pvm, szName, szEqn, hGloVarType);  

        break;  

  

    case CN_DYNAMICS:  

        if (iKWCode == KM_DXDT)  

            DefineDynamicsEqn (pibIn, szName, szEqn, ID_DERIV);  

        else  

            if (iKWCode == KM_FUNCTION)  

                DefineDynamicsEqn (pibIn, szName, szEqn, ID_FUNCTION);  

            else  

                DefineDynamicsEqn (pibIn, szName, szEqn, hGloVarType);  

  

        if (hGloVarType == ID_STATE && iKWCode != KM_DXDT)  

            ReportError (pibIn, RE_REDEF | RE_WARNING, szName,  

                         "Non-standard assignment in Dynamics section. "  

                         "Potential state discontinuity.\n");  

        break;  

  

    case CN_JACOB:  

        DefineJacobEqn (pibIn, szName, szEqn, hGloVarType);  

        break;  

  

    case CN_EVENTS:  

        DefineEventEqn (pibIn, szName, szEqn, hGloVarType);  

        break;  

  

    case CN_ROOTS:  

        DefineRootEqn (pibIn, szName, szEqn, hGloVarType);  

        break;  

  

    case CN_SCALE:  

        DefineScaleEqn (pibIn, szName, szEqn, hGloVarType);  

        break;  

  

    case CN_CALCOUTPUTS: /* FB added ID_NULL and RE_FATAL, 1 mar 98 */  

/* only outputs, local variables, and inlines can be assigned in  

CalcOutputs */

```

```

    if ((hGloVarType == ID_OUTPUT) ||
        (hGloVarType == ID_NULL) ||
        (hGloVarType == ID_LOCALCALCOUT) || /* added 7 mar 2008 */
        (iKWCode == KM_INLINE))
        DefineCalcOutEqn (pibIn, szName, szEqn, hGloVarType);
    else
        ReportError (pibIn, RE_BADCONTEXT | RE_FATAL, szName,
                      "Only outputs and local variables can be "
                      "defined in CalcOutputs{} section.");
    break;

    default:
        break;
} /* switch */

} /* DefineVariable */

/* -----
-----
DeclareModelVar

Declares szName to be a model variable of the type indicated
by iKWCode. Since the declaration commands States, Inputs, Outputs
and Compartments are only valid in the global context of the model
description file, it should be assured that this procedure is only
called in that context.

The number of States, Outputs, Inputs or Compartments is incremented,
and nominal value and defining equations strings are initialized to
NULL.
*/
void DeclareModelVar (PINPUTBUF pibIn, PSTR szName, int iKWCode)
{
    HANDLE hType;
    PINPUTINFO pinfo;

    pinfo = (PINPUTINFO) pibIn->pInfo;

    assert (iKWCode == KM_STATES || iKWCode == KM_INPUTS ||
            iKWCode == KM_OUTPUTS || iKWCode == KM_COMPARTMENTS);

    iKWCode = ((iKWCode == KM_STATES ? ID_STATE /* Translate to ID_ */
                                         : (iKWCode == KM_INPUTS ? ID_INPUT
                                         : (iKWCode == KM_OUTPUTS ? ID_OUTPUT
                                         : ID_COMPARTMENT))));

    if (!(hType = GetVarType (pinfo->pvmGloVars, szName))) { /* New id */
        if (iKWCode == ID_COMPARTMENT)
            AddEquation (&pinfo->pvmCpts, szName, NULL, (HANDLE) iKWCode);
        else
            AddEquation (&pinfo->pvmGloVars, szName, NULL, (HANDLE) iKWCode);
    }
}

```

```
else {
    if (hType == iKWCode) /* Same type redecl */
        ReportError (pibIn, RE_DUPDECL | RE_WARNING, szName, NULL);
    else {
        if (hType == ID_PARM) { /* Already init'd */
            ReportError (pibIn, RE_DUPDECL | RE_WARNING, szName,
                          "Model variable initialized before declaration");
            SetVarType (pinfo->pvmGloVars, szName, iKWCode);
        }
        else /* Unresolvable conflict */
            ReportError (pibIn, RE_DUPDECL | RE_FATAL, szName, NULL);
    }
}
} /* DeclareModelVar */
```